New features yield increase complexity

- Exciting Technology Advances
- Increased Design Challenges
- Increased Safety Concerns
Where do you *want* to discover and fix errors?

*Pre sales*  

*Model*  

*Generated Code*  

*Vehicle Testing*

*Post sales*  

*Vehicle (on road)*
Motivation

- It is easier and less expensive to fix design errors early in the process when they happen.
- Functional Safety Industry Standards ISO 26262

We will talk about ....

- How Model Based Design (MBD) facilitates early testing to incrementally increase confidence in your design
- Usage of V&V tools to enable delivery of higher quality software throughout the workflow
- Integration of V&V testing with code generation to improve confidence in the work product
Gaining Confidence in our Design

- Design error detection
- Req Linking MdI Advisor
- Functional & structural tests
- Code equiv & integration checks
- Field tests

Confidence vs. Effort / Time
Standards Context:
ISO 26262 “Road Vehicles - Functional Safety”

- Emerging functional safety standard for passenger cars
- Facilitates modern software engineering concepts such as
  - Model-Based Design
  - Early verification and validation
  - Code generation
- Provides objectives / requirements for
  - Development process activities (software safety life cycle)
  - Development and verification tools (tool qualification)
- First edition published 11th November 2011
ISO 26262 and Model-Based Design

Model-Based Design is deeply rooted in ISO 26262
### ISO 26262 Structure

| ISO 26262-1 | • Vocabulary |
| ISO 26262-2 | • Management of functional safety |
| ISO 26262-3 | • Concept phase |
| ISO 26262-4 | • Product development: system level |
| ISO 26262-5 | • Product development: hardware level |
| ISO 26262-6 | • Product development: software level |
| ISO 26262-7 | • Production and operation |
| ISO 26262-8 | • Supporting processes |
| ISO 26262-9 | • ASIL-oriented and safety-oriented analyses |
| ISO 26262-10 | • Guideline |

- Model-Based Design
- Early verification and validation
- Code generation
- Tool classification and qualification
ISO 26262 and Model-Based Design

- Model-Based Design is deeply rooted in ISO 26262
ISO 26262 and Model-Based Design

- ISO 26262-6 enables code generation and early verification and validation

### Table 11 — Methods for the informal verification of software unit design and implementation

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Inspection of the software unit design</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1b Walkthrough of the software unit design</td>
<td>++</td>
<td>+</td>
<td>O</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>1c Model Inspection&lt;sup&gt;a&lt;/sup&gt;</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1d Model Walkthrough&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++</td>
<td>+</td>
<td>O</td>
<td>o</td>
<td>o</td>
</tr>
<tr>
<td>1e Inspection of the source code&lt;sup&gt;b&lt;/sup&gt;</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1f Walkthrough of the source code&lt;sup&gt;b&lt;/sup&gt;</td>
<td>++</td>
<td>+</td>
<td>O</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>

<sup>a</sup> In the case of model-based software development the software unit specification can be verified at the model level by applying Method 1c and 1d.

<sup>b</sup> In the case of model-based development with automatic code generation, methods for informal verification of the generated code can be replaced by automated methods and techniques if applicable.

**NOTE 4** For model-based development, software unit testing can be carried out at the model level followed by back-to-back tests between the model and the code. The back-to-back tests are used to ensure that the behaviour of the models with regard to the test objectives is equivalent to the automatically-generated code.
Gaining Confidence in our Design

Confidence

Effort / Time

- Design error detection
- Req Linking Mdl Advisor
- Functional & structural tests
- Code equiv & integration checks
- Field tests
Model Based Design Verification Workflow

Model Verification
Discover design errors at design time

- Perform ad-hoc simulation testing
- Link requirements to model and tests
- Perform functional testing
- Measure model coverage

Code Verification
Gain confidence in the generated code

- Enforce modeling standards with Model Advisor
- Find defects early with Design Error Detection
Code Verification in Model Based Design

Model Verification

Discover design errors at design time

- Trace from code to model with Code Generation Report
- Compare model coverage to code coverage with the integrated 3rd party tool

Code Verification

Gain confidence in the generated code

- Perform Equivalence Testing using the integrated SIL mode
- Check the integrated code for run-time errors
Agenda

- Gaining confidence in your design with traditional functional testing and coverage
- Uncovering hard-to-find design errors like dead logic, divide by zero and integer overflows
- Gaining confidence in your design using model based formal testing methods
  - Prove implementation will meet requirements
  - Minimize the possibility of unintended behavior
  - Reproduce field issues and identify failure modes

The products shown will include

- **Simulink Verification and Validation** – Model Coverage, Req Linking, Model Advisor
- **Simulink Test** – Functional Testing, Test Manager
- **Simulink Design Verifier** – Design Error Detection, Property Proving, Input Generation
- **Polyspace Code Prover** – Property Proving, Range Analysis, MISRA checking
Example

Informal testing:

Ad-hoc “Eyeball” Testing
Ad-hoc Testing

New “Dashboard” blocks facilitate early ad-hoc testing
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?
- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Gaining Confidence in our Design

- **Confidence**
  - **Effort / Time**
    - Design error detection
    - Functional & structural tests
    - Code equiv & integration checks
    - Field tests
  - Req Linking Mdl Advisor

---

MathWorks®

AUTOMOTIVE CONERENCE 2015
Example

Static checks:

Requirements Linking
Model Advisor
Requirements Linking – Highlighting Objects with Links

1. Vehicle braking will transition system to disengage
   when engaged (active)
   The cruise control system shall transition to disengaged from engaged
   when a braking event has occurred.

2. Transition to disengaged (inactive) when vehicle s...
   The cruise control system target speed shall transition to disengaged when
   the current vehicle speed is outside the limits of 20 to 90 mph.

![Diagram of cruise control system with highlighted objects and links]
1.4. Engaged (active) Output
The controller shall have an output signal to indicate that the controller is engaged (active).

1.5. Target Speed Output
The controller shall have an output signal to indicate that the target speed of the controller.

1.6. Vehicle Speed Input
The controller shall have a vehicle speed input to be used by the target speed algorithm.

1.7. Vehicle Brake Input
The controller shall have a vehicle brake input to indicate when the driver has applied the brake pedal.
Model Advisor – Model Standards Checking
Enforcing Modeling Standards: Model Advisor Features

- Static analysis of models against a set of checks
  - Checks for simulation configuration
  - Checks for code generation configuration
  - Requirements consistency
  - Modeling Standards

- Modeling Standards Checks (SLVV)
  - MAAB Style Guidelines
  - DO-178B
  - IEC 61508 and ISO 26262 checks

- API for process customization

- Create custom checks and configurations
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?
- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?
- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Gaining Confidence in our Design

Confidence

Mdl Advisor
Req Linking

Design error
detection

Functional & structural
tests

Code equiv & integration
checks

Field tests

Effort / Time
Example

Static checks:

Finding Design Errors:
Dead Logic
Simulink Design Verifier identifies “Dead logic” in model.

Finding Design Errors: Dead Logic

Simulink Design Verifier identifies “Dead logic” in model.
Finding Unintended Behavior

Converting floating-point model to integer calibrations, signals…

- Dead logic due to “uint8” operation
Finding Unintended Behavior

- **Dead logic** due to “uint8” operation on `incdec/holdrate * 10`

- **Fix** change the order of operation
  
  \[10 \times incdec / holdrate\]

```
debug>> incdec
incdec = 1
debug>> holdrate
holdrate = 5
debug>> incdec/holdrate
ans = 0
```

Condition can never be false
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?
- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?
- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Gaining Confidence in our Design

- Design error detection
- Functional & structural tests
- Code equiv & integration checks
- Field tests

Confidence vs. Effort / Time

MathWorks AUTOMOTIVE CONFERENCE 2015
Simulation Testing Workflow

Requirements

Did we meet requirements?

Review functional behavior

Did we completely test our model?

Structural coverage report

Design

Functional

Structural
Did We Completely Test our Model?

Model Coverage Analysis

Potential causes of less than 100% coverage:

- Missing requirements
- Over-specified design
- Design errors
- Missing tests
Model Coverage: What is Model Coverage?

Decision Coverage (DC)

Percentage of paths taken through decision point

if \((X \& Y)\)
\[ Z = 1; \]
else
\[ Z = -1; \]
end
Model Coverage: What is Model Coverage?

Condition Coverage (CC)

Percentage of conditions exercised

if (X & Y)
  z = 1;
else
  z = -1;
end
Model Coverage:
What is Model Coverage?

Modified Condition/Decision Coverage (MC/DC)

Checks how inputs independently affect output

Affects (X & Y) to be T and F?
Affects (X & Y) to be T and F?

if \((X \& Y)\)
\begin{align*}
\text{if } & (X \& Y) \\
\text{z} & = 1; \\
\text{else} \\
\text{z} & = -1; \\
\text{end}
\end{align*}
Example

Model testing before code generation:

Requirements Based Functional Testing with Coverage Analysis
All 14 requirements based test cases pass

By analyzing model coverage results we find:
- Missing test cases for vehicle speed exit conditions,
- Missing requirements (and test cases) for “hold” or continuous speed button input
Functional Testing with Added Requirements & Test Cases

- Added 2 new requirements for the “hold” case for speed setting input buttons
- Added 5 test cases to the original 14 requirements based test cases
  - 3 test cases for the 2 new requirements
  - 2 test cases for the missing test cases for the vehicle speed exist conditions
- 4/5 new functional test cases pass
  - Failed test case showed overshoot beyond target speed limits
  - Coverage analysis highlighted transitions with design errors
  - Fixed comparison operators, (<) → (<=), and (>) → (>=)

- Now all (19) functional test cases pass with 100% model coverage!
## Simulink Test Major Features

<table>
<thead>
<tr>
<th>Test Manager</th>
<th>Test Harness</th>
<th>Test Sequence Block</th>
</tr>
</thead>
</table>
| • Author, execute, manage test cases  
  • Review, export, report | • Synchronized, simulation test environment | • Test Inputs and assessments  
  • Based on logical, temporal conditions |

![Test Manager](image1)

![Test Harness](image2)

![Test Sequence Block](image3)
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?
- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?

- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Gaining Confidence in our Design

- Code equiv & integration checks
- Field tests
- Functional & structural tests
- Design error detection
- Req Linking Mdl Advisor

Confidence vs Effort / Time
Equivalence Testing: Model vs SIL or PIL Mode Testing

Coverage $\rightarrow 100\%$

Model Testing

- Model used for production code generation
  - Test vectors $i(t)$
  - Simulation
  - Result vectors (base line) $o_{sim}(t)$

SIL or PIL Mode Testing

- Embedded Coder
- Generated C code
- Target compiler and linker
- Object code
- Execution
- Result vectors $o_{code}(t)$
- Signal comparison
Example

Testing of the generated code:

*Test Generation for Full Coverage*

*Equivalence Testing*

*Model vs Code Coverage Comparison*
Traceability: Code Generation Report

Rich_Mixture entry: fuel_mode = RICH;

Single_Failure
Test Vector Generation for Equivalence Testing: Adding to Requirements Based Test Coverage
Code Equivalence Check Results: Model vs Code

- Re-used full coverage test vectors and harnesses from Model Verification testing
- Ran test vectors on generated code using Model Reference SIL mode
- Equivalence test performed in Simulink Test, including test execution, evaluation and presentation of the results
- Compared Model Coverage to Code Coverage using the integrated support for a code coverage tool in the Code Generation Report
- Successfully demonstrated code behavior matches model behavior!
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?

- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?
- Does the generated code behave the same as the model?

Can we prove the implementation always meets requirements?
Gaining Confidence in our Design

- Design error detection
- Functional & structural tests
- Code equiv & integration checks
- Field tests

Issues were identified and fixed with each test but …

Field testing will show the algorithm still had issues
Gaining Confidence in our Design

Confidence

--

Effort / Time

- Design error detection
- Functional & structural tests
- Code equiv & integration checks
- Field tests

Req Linking
Mdl Advisor
Field Test Results:
Has something like this happened to you?

From: Larry Findebug
Sent: Monday, March 2, 2015 9:29 PM
To: Chuck Olosky
Subject: Cruise Control Target Speed Issue

I ran into an unexpected issue with the new Cruise Control feature during vehicle testing.

**While going downhill, random target speed increases with “reduce speed” button**
- a) I set the target speed while going downhill on the track going around 40 mph
- b) While pulsing the “reduce speed” button, target speed reduced to 20 mph limit
- c) Next time I hit “reduce speed”, it increased the target speed from 20 mph to 33 mph
- d) Another time it increased the target speed from 20 mph to 29 mph

Hopefully we can fix this issue quickly. These issues that don’t repeat themselves can be hard to find. Let me know if I can help.  ---Larry F

*While going downhill, target speed increases with “reduce speed” button*
Field Tests Uncover Issue

**Problem:** While going downhill, target speed increases with “reduce speed” button and assumes random values

- Functional tests pass for model
- Completely tested the model (100% coverage achieved)
- Nominal signal and parameter values worked in simulation
How can debug the issues found in the field?

- Reproduce field issue in Simulink
  - Use field calibration values
  - Set inputs manually or replay data if it exists

- But this is just “one” more test case that covers a subset of all possible input ranges, sequences and calibration values

- Already have requirements based test vectors, 100% coverage, and all passed

- How do I know that this will not happen with a different set of conditions?
  - Road conditions
  - Driver inputs
  - Calibration values
  - Sequence of events
- Construct a model of field issue for Simulink Design Verifier
- Constrain inputs to represent field issue
- Create model of field issue behavior
- Ask tool to prove whether errant condition can occur
Field Issue Behavior Model
Target speed increases with “reduce speed” button

a) I set the target speed to the vehicle speed (40 mph) while going downhill on the track

b) I was pulsing the “reduce speed” button until it decreased the target speed to the 20 mph limit

c) The next time I hit the “reduce speed” button it increased the target speed from 20 mph to 33 mph
Example

Formal verification with property proving:

Reproducing Field Issues

Requirements “property” models

Checking more conditions
Reproducing Field Issues

- Recreated the field issue
  - Constrained inputs to match field inputs
  - Modeled field issue behavior as a property
- Property Proving generated a test case that reproduced the field issue
- Generated harness was used to debug and demonstrate the fix
Proving Model will Meet Requirements

- Modeled requirement as a Property model
  - Unconstrained inputs
  - Modeled intended requirement behavior

- Property Proving generated a test case that violated the requirement

- Found unintended behavior when both “speed change” buttons pressed

- Added algorithm to reject double press inputs

- Proved model now meets this requirement!
Re-run Functional Tests after Property Proving Fixes

- Partial model coverage for “RejectDoublePress”
- Requirements and test cases need to be updated
Property Proving Work Summary

- Use “Property Proving” to prove the algorithm will always perform with the intended behavior
  - Construct the Property Proving Model to reflect the requirement or field issue
  - Model the intended or observed behavior
  - Constrain the signal inputs per requirement or field issue
  - Sweep or constrain the calibrations range

- If property is falsified (disproved)
  - Use the Design Verifier generated harness to debug issue, then discard harness
  - Re-run Property Proving to verify the fix

- In the example, we showed an iterative workflow to improve confidence
  - “Sneak path” due to incomplete refactoring of the model
  - A more general “reduce speed” property showed a simultaneous input issue
  - Sweeping with all possible calibration values showed a limit check issue
Increasing Confidence beyond Ad-hoc Testing

Ad-hoc testing is a very useful method but what else do we need to check?

- Are all requirements implemented?
- Does the implementation follow modeling standards?
- Have we shown the absence of design errors (dead logic, divide by zero, overflow)?
- Does the implementation pass all functional requirements?
- Do the tests completely exercise the implementation?
- Does the generated code behave the same as the model?
- Can we prove the implementation always meets requirements?
Increasing Design Confidence with Model and Code Verification

Stefan David – Application Engineering @ MathWorks Germany

MathWorks
AUTOMOTIVE CONFERENCE 2015
Why code verification? Challenges

Vehicles recalled (in millions)

Source: National Highway Traffic Safety Administration
How to address these challenges? Derived requirements

- MISRA-C
- Comply with ISO, IEC safety requirements
- Optimize Design and Architecture
- Deliver Bug free software
- Measure Code Quality
- Reduce time and efforts for software testing
Example

Formal verification at the code level:

Detect software integration defects earlier

Prove code robustness

Optimize design decisions
Why code verification? Typical Embedded Systems Architecture

Controller Model

Generated Algorithm Code

Input Drivers

Output Drivers

Comm Drivers

Special Device Drivers

Scheduling/Operating System and Support Utilities

Communication Interfaces

Sensors

Special Interfaces

Actuators

Scheduler/Operating System and Support Utilities

Included Target Optimized and Legacy Code
Why code verification? Typical ECU System Architecture

**Inputs**
- Cruise_onoff
- Brake
- Speed
- Coast set
- Accel reset
- EGO Sensor
- MAP Sensor

**ECU system**
- (e.g. AUTOSAR RTE)

**Outputs**
- Gear
- Engaged
- Target speed
- Fuel Rate

**Legacy code**

- Cruise Control Module (MBD)
- Fuel Rate Control Module
- Shift Logic Control Module

[Image of diagram with various inputs, ECU system, and outputs labeled]
Why code verification? Detect Code Integration Issues

Target speed parameter propagated to “Cruise_ctrl.c” [0 … 40]

Maximum target speed = 90

Dead code

```c
/* Entry 'STANDBY': '<S5>:52' */
*rty_Engaged = false;
} else if (rtu_Speed > maxtspeed) {
    /* Transition: '<S5>:55' */
    /* Exit Internal 'ON': '<S5>:54' */
    localDW->is_ON = IN_NO_ACTIVE_CHILD;
    localDW->is_CRUISE = IN_STANDBY;
    /* Entry 'STANDBY': '<S5>:52' */
    *rty_Engaged = false;
} else if (rtu_Speed < mintspeed) {
    /* Transition: '<S5>:113' */
```
Search for Parameter in Upstream Source
Use Call Graphs for Multi-File Traceability
Root Cause for Dead Code

Changing analog-to-digital converter from 14 to 12-bit results in dead code

12bits max = 4095 value. Multiply by pre-existing scaling of 0.01 = 40.95. My speed will saturate at 40mph

MASK – accounts for scaling down for new ADC from 14-bit to 12-bit

CONV_FACTOR – accounts for translating sensor input to miles/hr

Overlooked changing CONV_FACTOR for new ADC
Find Dead Code During Integration

Inputs
- Cruise_onoff
- Brake
- Speed
- Coast set
- Accel reset
- EGO Sensor
- MAP Sensor

Outputs
- Gear
- Engaged
- Target speed
- Fuel Rate

Inputs:
- Inaccurate scaling for speed

Outputs:
- Cruise Control Module (MBD)
- Fuel Rate Control Module
- Shift Logic Control Module

Legacy code

Dead code
Model-Based Design Workflow

<table>
<thead>
<tr>
<th>Initiation</th>
<th>Architecture</th>
<th>Design/Implementation</th>
<th>Unit Testing</th>
<th>Integration Testing</th>
</tr>
</thead>
</table>

**Requirements**
- System-level specification
- Architectural design: environment models, physical components, algorithms
- Subsystem design
- Component design: test vectors and expected responses

**Architectural Design**
- Environment models
- Physical components
- Algorithms

**Component Design**
- Test vectors and expected outputs

**Implementation**
- C, C++
- MCU, DSP
- VHDL, Verilog
- Structured text: FPGA, ASIC, PLC

**System Tests**
- System tests in simulation
- Run system tests on integrated controller

**Component Tests**
- Component tests in simulation
- Run component tests on target

**System validation/acceptance Tests**
- User acceptance testing

**Integration Testing**
- Complete integration & test
- System-level integration & test

**Subsystem Integration & Test**
- Subsystem design
- Subsystem implementation

**Subsystem Design**
- Test vectors and expected responses

**Early Verification**
- Test vectors and expected outputs

**Notes**
- Code verification and validation
- System tests in simulation
- Run system tests on integrated controller
- Subsystem integration & test
- Subsystem implementation
Why code verification? Tests aren’t exhaustive!

“Program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra [1])

Why code verification? Code verification is exhaustive!

Statically and semantically verifies all possible executions of your code (considering all possible inputs, paths, variable values)

- Finds **runtime errors**, **boundary conditions** and **unreachable code** without testing
- **Proves** when code **will not fail** under any runtime conditions, reducing robustness testing efforts
- Gives insight into runtime behavior and data ranges

```
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
        *p = 0;
        p++;
    }

    if (get_bus_status() > 0) {
        if (get_oil_pressure() > 0) {
            *p = 5;
        } else {
            i++;
        }
    }

    i = get_bus_status();

    if (i > 0) {
        *p = i * 10;
    }
}
```
When using Code Verification?
Example Workflow(s)

Specification
- Check Model to code conformance
- Find robustness issues
- Find variable scaling issues
- Optimize generated code size

Design
- Review
  - Find local bugs
  - Find and justify MISRA violations
  - Find “untestable” functions

Implementation
- SW Acceptance tests
  - Create quality reports
- SW Integration tests
  - Find integration bugs
    - Declaration mismatches
    - Data race on shared variables
    - Global variable usage
- SW Unit Tests
  - Quality gate
    - Find runtime errors / unused code
    - Prove robustness of modules
    - Detect unused/dead code
Polyspace product family for C/C++

- **Polyspace Code Prover**
  - Proves code to be safe and dependable
  - Deep verification of software components
  - Perform QA signoff for production ready code

- **Polyspace Bug Finder**
  - Quickly find bugs in embedded software
  - Check code compliance for MISRA and JSF
  - Intended for every day use by software engineers

Ada language also supported for proving code
Run-Time Errors and Software Defects

- Non-initialized data
- Out of bound array access
- Null pointer dereference
- Incorrect computation
- Concurrent access to shared data
- Illegal type conversion
- Unreachable states or code
- Overflows
- Non-terminating loops
- And lots more…

- Invalid use of = operator
- Invalid use of == operator
- Write without further read
- Uncalled function
- Missing null in string array
- Qualifier removed in conversion
- Race condition
- Invalid use of other standard library routine
- Memory leak
- And lots more…
Use Case: Optimize design and architecture decisions

Non Robust Module

External code

Potential Runtime Error inside!!!
Use Case: Optimize design and architecture decisions

Non Robust Module

Additional Range-Limiting Code

Free from Runtime Errors
Use Case: Optimize design and architecture decisions

Code Generation

Optimization Settings

Optimized code AND robustness
Use Case: Coding guidelines checking

- MISRA-C:2012
  - ALL rules supported
  - Several directives supported

- MISRA-C:2004
  - 131 rules supported

- MISRA AC AGC
  - 88 obligatory rules are supported

- MISRA-C++:2008
  - 185 of the 228 rules supported

- JSF++:2005
  - 157 of 234 rules supported

- Customization
  - Turn rules off / warning / error
  - Define custom naming conventions
Challenge: MISRA vs. Effort for changing code

Hint: Analysis accuracy is specifically valuable for generated and legacy C code

MISRA Violation

Proven save (no Overflow)

Justify MISRA violation

Polyspace Code Prover
Advantage: Leverage an integrated tool chain
For Model-Based Design and automatic code generation and hand-written code

- Context information for inputs and parameters
- Traceability back to the model
- Block-level annotations to filter warnings
Use Case: Coverage of ISO 26262 requirements (*)

**Polyspace Bug Finder**

- [PBF_UC1] Static analysis of C/C++ code to assess compliance with coding standards
- [PBF_UC2] Static analysis of C/C++ code to determine code size and complexity metrics
- [PBF_UC3] Determination of software quality metrics
- [PBF_UC4] Static analysis of C/C++ code to assess interface between components
- [PBF_UC5] Static analysis of C/C++ code to detect systematic and potential software defects

**Polyspace Code Prover**

- [PCP_UC1] Semantic code analysis with abstract interpretation of C/C++ Code to detect systematic and potential run-time errors
- [PCP_UC2] Semantic code analysis with abstract interpretation of C/C++ code to detect unreachable code
- [PCP_UC3] Semantic analysis of the calling relationships in the C/C++ code
- [PCP_UC4] Semantic analysis of global variable usage in the C/C++ code
- [PCP_UC5] Reporting of software quality metrics
- [PCP_UC6] Semantic analysis of C/C++ code to assess interface between components

*) ISO 26262-8 pre-qualified use cases
Use Case: Freedom from interference (ISO 26262-6 Annex D)

- **Goal:** Prevent or detect faults that can cause interference between SW elements (e.g. different SW partitions)

- **Polyspace can partly prevent or detect faults:**
  - **D2.2 Timing and execution**
    - Deadlocks (BF)
    - Double locks (BF)
    - Missing unlocks (BF)
    - Data Race Conditions (BF)
  - **D2.3 Memory**
    - Corruption of content
      - Out-of-bound access (BF, CP)
    - Read or write access to memory allocated to another software element
      - Exhaustively identify unprotected shared variables (CP)
      - Document read-/write access to global variables (CP)
What's New in Polyspace R2015b – Many new defects detected

- 80+ checkers added in Bug Finder R2015b
- New security defect and tainted data checks
- Instant results
- Full MISRA-C:2012
- ...

Ford Powertrain Presentation:
What’s New in Polyspace – Detect Security Vulnerabilities

- TBD
- Embed 2Min Video:
Why code verification? Challenges

- Check MISRA-C
- Comply with ISO, IEC safety requirements
- Optimize Design and Architecture
- Deliver Bug free software
- Measure Code Quality
- Reduce time and efforts for software testing
Summary: Leverage productivity gain of range analysis!
For Model-Based Design and automatic code generation and hand-written code

- Reduce effort for tests AND have confidence in generated code
- Optimize generated code size AND have robust modules
- Satisfy MISRA-C AND maintain high code performance

...and of course detect a number of defects early!
Gaining Confidence in our Design

- **Functional & structural tests**
  - Tests entire specified range of input signals and calibrations
  - Ensures implementation will always meet requirements
  - Generates a test case upon failure for debugging
  - Increases design confidence

- **Design error detection**

- **Req Linking Mdl Advisor**

- **Requirement proofs**

- **Code integration checks**

- **Field tests**
Key Enablers

*MathWorks tools enable* ...

- The usage of a standard workflow for software development
- A seamless, integrated workflow for model development, code generation and V&V testing
- Design verification earlier in the development lifecycle and support across the lifecycle
- Increased confidence in the work product with integrated support for code verification
Thank You!
Thanks! What’s Next?

Visit the Model and Code Verification Demo Pod to learn more!